# データとコードの並べ替え: 最適化とメモリー - パート 2

この記事は、インテル® デベロッパー・ゾーンに公開されている「Putting Your Data and Code in Order: Data and layout - Part 2」の日本語参考訳です。

---

パフォーマンスとメモリーに関するこのシリーズ (全 2 パート) では、ソフトウェア・パフォーマンスを向上したいと考えている開発者にそのためのガイダンスとなる基本的な概念を提供します。このシリーズでは特に、メモリーとデータレイアウトについて考察します。パート 1 では、レジスターの使用とタイリング/ブロッキング・アルゴリズムによりデータの再利用を改善しました。パート 2 では、最初に、一般的な最適化 (スレッドを使用する共有メモリー・プログラミング) 向けのデータレイアウトについて取り上げ、次に MPI を利用する分散コンピューティングについて見ていきます。そして、それらの概念を拡大して、並列性、ベクトル化 (SIMD: Single Instruction Multiple Data) および共有メモリー並列処理 (スレッド化)、分散メモリー・コンピューティングに触れ、最後に、構造体配列 (AoS) と配列構造体 (SoA) データレイアウトについて考えます。

パート 1 で強調したパフォーマンスの基本原則は、レジスターやキャッシュから退避される前にデータをできるだけ再利用するということです。パート 2 で強調するパフォーマンスの原則は、最もよく使用される場所に隣接してデータを配置する、連続アクセスモードでデータを配置する、そしてデータ競合を回避することです。

## スレッドを利用する共有メモリー・プログラミング

最初に、スレッドを利用する共有メモリー・プログラミングについて考えてみます。すべてのスレッドは同じプロセスに属し、メモリー空間を共有します。多くのスレッド化モデルがありますが、最もよく知られているのは Posix* スレッドと Windows* スレッドです。スレッドの生成と管理を適切に行うための作業は面倒です。現代のソフトウェアは、多数のモジュールで構成され、大きな開発チームによって作成されるため、スレッドを利用する並列プログラミングではエラーが発生しがちです。スレッドの生成、管理、そして並列スレッドを最大限に活用するのを支援するため、いくつかのスレッド化パッケージが開発されました。最もよく使用されているのは、OpenMP*、インテル® スレッディング・ビルディング・ブロック (インテル® TBB)、インテル® Cilk™ Plus ですが、インテル® Cilk™ Plus は OpenMP* やインテル® TBB ほど普及していません。これらのスレッド化モデルは、スレッドプールを作成し、並列処理や並列領域で再利用します。OpenMP* は、ディレクティブを使用したインクリメンタルな並列化が可能です。通常、OpenMP* ディレクティブは、段階的に最小限のコード変更で既存のソフトウェアに追加することができます。OpenMP* ランタイム・ライブラリーにスレッドの管理のほとんどを任せることで、ソフトウェアのスレッド化を容易に行うことができます。さらに、すべてのコード開発者に一貫したスレッド化モデルを提供し、一般的なスレッドエラーを軽減し、スレッドを最適化するため開発者によって作成された最適化されたマルチスレッド・ランタイム・ライブラリーを提供することができます。

冒頭で述べた並列化の基本原則は、データを使用される場所の近くに配置し、データの移動を回避することです。スレッド・プログラミングのデフォルトのモデルでは、データはプロセスでグローバルに共有され、すべてのスレッドがアクセスする可能性があります。スレッド化を紹介する多くの記事では、do ループ (Fortran) や for ループ (C) に OpenMP* を適用することで簡単にスレッド化できることを強調しています。これらの手法は、通常、2 ~ 4 コアで実行する場合によくスピードアップします。多くの場合、64 スレッド以上までスケーリングしますが、スケーリングしないこともあります。その場合、データ分割が効率的に行われていない可能性があります。つまり、並列コードの構造に問題があります。

開発者やソフトウェア・ツールによって特定された並列化可能な場所ではなく、コールスタックのより上位のレベルで並列化を実装することが重要です。並列に実行できるタスクやデータを見つけた場合、開発者はアムダールの法則に照らし合わせて、「この場所よりもコールスタックの上位で並列処理を開始できるか? この並列化により、コードの並列領域が増加し、スケーラビリティーが向上するか?」を問うべきです。

データの配置とメッセージを介して共有されるデータについてよく検討する必要があります。最もよく使用される場所にデータが配置され、必要に応じて別のシステムに送られるようなデータレイアウトにすべきです。グリッドで表現されるアプリケーションや特定のパーティションを持つ物理ドメインの場合、MPI ソフトウェアはよくサブグリッドやサブドメインの周囲に「ゴースト」セルを含む行を追加します。ゴーストセルは、それらのセルを更新する MPI プロセスから送られたデータの値を格納するのに使用されます。一般に、マルチスレッド・ソフトウェアではゴーストセルは使用されませんが、メッセージパッシングのパーティションのエッジの長さを最小化するのと同様に、共有メモリーを使用してスレッドのパーティションのエッジも最小化したほうが良いでしょう。そうすることで、スレッドロック (クリティカル・セクション) またはアクセス権限に伴うキャッシュの利用ペナルティーを最小限に抑えることができます。

大規模なマルチソケット・システムでは、グローバル・メモリー・アドレス空間を共有しているにもかかわらず、通常 NUMA (Non-Uniform Memory Access) 時間の影響を受けます。別のソケットに近いメモリーバンクにあるデータのほうが、コードを実行しているソケットに近いバンクにあるデータよりも取得に時間がかかります (レイテンシーが長くなります)。近くのメモリーのほうが、短いレイテンシーでアクセスできます。

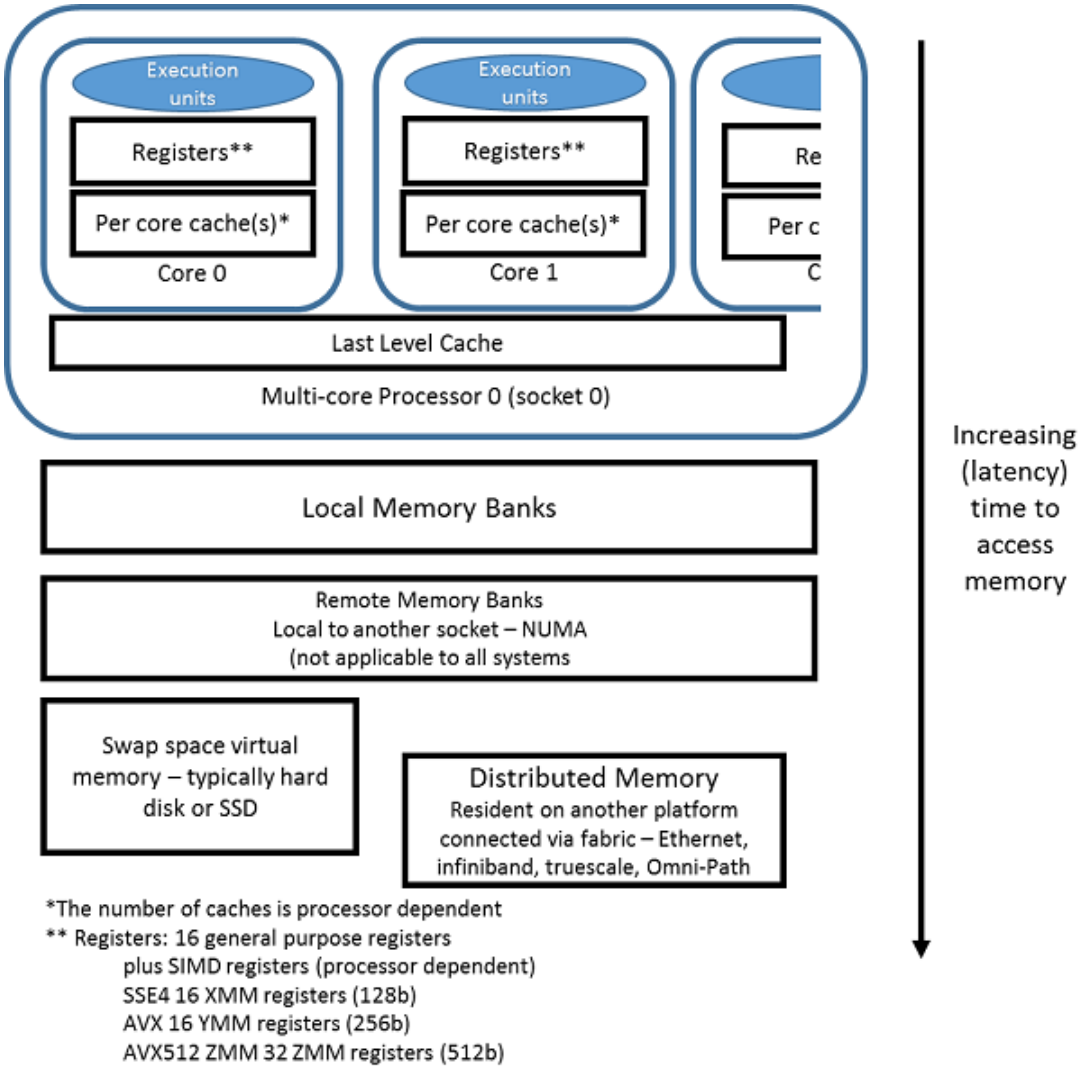



図 1. メモリーアクセスのレイテンシー (データアクセスの相対時間)

1 つのスレッドがデータの割り当てと初期化を行う場合、通常、そのスレッドが実行しているソケットに最も近いバンクにデータが配置されます (図 1)。各スレッドが主に使用するメモリーを割り当てて、最初に参照することでパフォーマンスを向上できます。多くの場合、これだけでスレッドが実行しているソケットに最も近いメモリーを利用できます。スレッドが生成されアクティブになると、OS は通常スレッドを 1 つのソケットに配置します。場合によっては、スレッド・マイグレーションを回避するため、明示的にスレッドを特定のコアにバインドしたほうがよいことがあります。データに特定のパターンが見られる場合、そのパターンに一致するように、スレッドを特定のコアに割り当て、バインド、またはアフィニティー設定することで利点が得られます。インテルの OpenMP* ランタイム・ライブラリー (インテル® Parallel Studio XE に含まれる) は、インテル® Xeon Phi™ プロセッサー/コプロセッサーで役立つ明示的なマップ属性 (compact、scatter、および balanced) を提供します。

- compact 属性は、対称型マルチスレッディング (SMT) の連続または隣接するスレッドを単一コア上に割り当ててから、別のコアにスレッドを割り当てます。これは、スレッドが連続する番号の (隣接する) スレッドとデータを共有する場合に理想的です。

- scatter 属性は、各コアにスレッドを 1 つずつ割り当てて、最後のコアに到達したらまた最初のコアから順に割り当てます。
- balanced 属性は、連続または隣接する ID のスレッドを同じコアにバランスよく割り当てます。インテル® C++ コンパイラーのドキュメントに従ってスレッド・アフィニティーを最適化する場合、balanced 属性から開始することを推奨します。balanced 属性は、インテル® Xeon Phi™ 製品ファミリーでのみ利用可能で、一般的な CPU では無効です。インテル® Xeon Phi™ 製品ファミリー・ベースのプラットフォーム上ですべての SMT を利用する場合、balanced と compact の動作は同じになります。インテル® Xeon Phi™ 製品ファミリー・ベースのプラットフォーム上で一部の SMT のみ利用する場合、compact はすべての SMT を最初のほうのコアに割り当て、最後のほうのコアはアイドル状態になります。

多くのスレッドを使用する場合、スレッドデータを使用される場所の近くに配置することが重要です。MPI プログラムと同様に、スレッド化されたプログラムでもデータレイアウトは重要です。

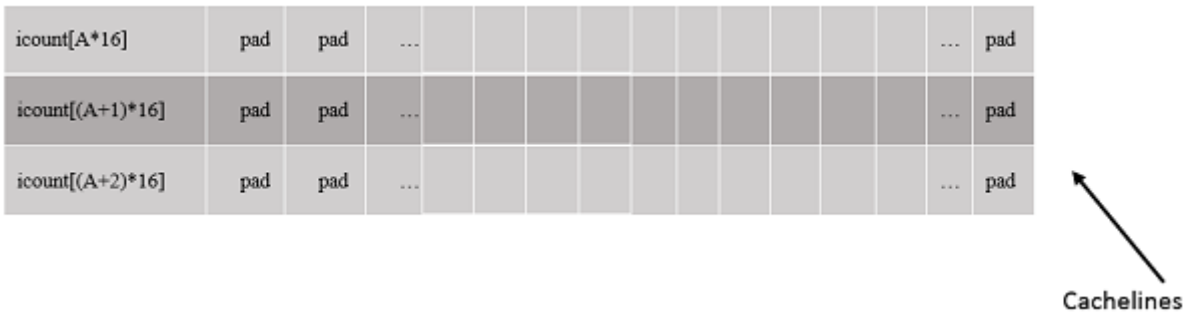
メモリーとデータレイアウトに関して、2 つの項目を考慮すべきです。それらは比較的簡単に対応することができ、大きな効果が得られます。1 つ目はフォルス・シェアリングで、2 つ目はデータ・アライメントです。スレッド化されたソフトウェアにおける興味深いパフォーマンスの問題の 1 つにフォルス・シェアリングがあります。各スレッドが操作するデータは独立しており、共有されていませんが、それぞれのデータが同じキャッシュライン上に存在します。そのような状況をフォルス・シェアリング (または、フォルス・データ・シェアリング) と呼びます。データは共有されていませんが、パフォーマンス上は共有されているかのように振る舞います。

各スレッドが 1 次元配列に格納されているそれぞれのカウンターをインクリメントする場合について考えてみます。カウンターをインクリメントするには、コアがキャッシュラインを所有していなければなりません。例えば、ソケット 0 のスレッド A はキャッシュラインの所有権を取得し、iCount[A] をインクリメントします。一方、ソケット 1 のスレッド A+1 は iCount[A+1] をインクリメントします。そのためには、ソケット 1 のコアがキャッシュラインの所有権を取得して、スレッド A+1 がその値を更新します。キャッシュラインの値が変更されるため、ソケット 0 のプロセッサーのキャッシュラインが無効化されます。次の反復で、ソケット 0 のプロセッサーがソケット 1 からキャッシュラインの所有権を取得し、iCount[A] の値を変更します。これにより、ソケット 1 のキャッシュラインが無効化されます。ソケット 1 のスレッドが書き込みを行う際にも、同様のサイクルが繰り返されます。キャッシュの一貫性を保持するため、キャッシュラインの無効化、所有権の再取得、メモリーとの同期に多くのサイクルが費やされ、パフォーマンスに影響します。

この問題に対する最良の解決策は、キャッシュラインが無効化されないようにすることです。例えば、ループの入り口で各スレッドがそれぞれのカウントを読み取り、スタック上のローカル変数に格納します (読み取りはキャッシュを無効化しません)。そして、作業完了後にローカル変数の値をキャッシュにコピーします (図 2)。別の方法は、特定のスレッドで主に使用されるデータのみがそのキャッシュラインに配置されるように、データにパディングを追加します。

```
int iCount[nThreads] ;
    .
    .
    .
    for (some interval){
        // 処理...
        iCount[myThreadId]++ // フォルス・シェアリングの可能性
    }
```


```
int iCount[nThreads*16] ; // フォルス・シェアリングを回避するためのメモリーパディング
    .
    .
    .
    for (some interval){
        // 処理...
        iCount[myThreadId*16]++ // フォルス・シェアリングではない、未使用のメモリー
    }
```




```
int iCount[nThreads] ; // 一時ローカルコピーを作成
    .
    .
    .
    // スレッドごとに個別のローカル変数 local_count を作成
    int local_Count = iCount[myThreadID] ;
    for (some interval){
        // 処理...
        local_Count++ ; // フォルス・シェアリングなし
    }
    iCount[myThreadId] = local_Count ; // 値を保持
    // 最後にフォルス・シェアリングが発生する可能性があるが
    // 内部作業ループの外側が大幅に向上
    // スレッドごとに local_Count を保持するほうがよい
```

図 2.

フォルス・シェアリングは、隣接するメモリー位置に割り当てられたスカラーでも発生することがあります。以下にコード例を示します。

```
int data1, data2 ; // data1 と data2 はフォルス・シェアリングが
                   // 発生する可能性がある方法でメモリーに配置されている
declspec(align(64)) int data3; // data3 と data4 は
declspec(align(64)) int data4; // 別々のキャッシュライン上にあるため
                               // フォルス・シェアリングは発生しない
```

開発者が並列処理をゼロから設計し、共有データの使用を最小限に抑える場合、一般にフォルス・シェアリングは回避できます。スレッド化したソフトウェアがうまくスケーリングしない場合、独立した処理が多く、いくつかのバリアー (mutex、クリティカル・セクション) を使用していても、フォルス・シェアリングのチェックを行うとよいでしょう。

## データ・アライメント

ソフトウェアのパフォーマンスは、SIMD 処理 (インテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512)、インテル® AVX、インテル® ストリーミング SIMD 拡張命令 4 (インテル® SSE4)、...) で扱うデータがキャッシュライン境界でアライメントされている場合に最適になります。アライメントされていないデータアクセスのペナルティーは、プロセッサー・ファミリーによって異なります。インテル® Xeon Phi™ コプロセッサーは、特にデータ・アライメントの影響を受けやすいため、インテル® Xeon Phi™ コプロセッサー・ベースのプラットフォームでは、データ・アライメントが非常に重要です。インテル® Xeon® プロセッサー・ベースのプラットフォームほど顕著な違いは見られないものの、データがキャッシュライン境界でアライメントされている場合、パフォーマンスが大幅に向上します。このため、常に 64 バイト境界でアライメントすることを推奨します。Linux* および macOS* では、ソースコードを変更せずに、インテル® コンパイラーの `/align:rec64byte` オプションを使用するだけでアライメントを調整することができます。

C コードで動的に割り当てられるメモリーでは、`malloc()` を `_mm_alloc(`*datasize,* `64)` に置換します。`_mm_alloc()` を使用する場合、`free()` の代わりに `_mm_free()` を使用します。データ・アライメントの詳細は、https://www.isus.jp/products/c-compilers/data-alignment-to-assist-vectorization/ を参照してください。

コンパイラー・ドキュメントも参考になります。データ・アライメントの効果を確認するため、同じサイズの 2 つの行列を作成して、パート 1 のブロック化した行列乗算コードを実行しました。行列 A をアライメントしたケース 1 と行列 A を意図的に 24 バイト (3 つの倍精度) でオフセットしたケース 2 を用意し、インテル® コンパイラー 16.0 でコンパイルして、行列サイズ 1200x1200 ~ 4000x4000 で実行したところ、パフォーマンスが 56% ~ 63% 低下しました。パート 1 では、異なるコンパイラーによるループの順序のパフォーマンスを表に示しました。1 つの行列がオフセットされている場合、インテル® コンパイラーを使用してもパフォーマンスの利点は得られません。データがアライメントされている場合にコンパイラーがその情報を有効に活用できるように、コンパイラー・ドキュメントを参照してデータ・アライメントと利用可能なオプションについて確認して

ください。キャッシュラインから行列オフセットのパフォーマンスを評価するコードは、パート 1 のコード (https://github.com/drmackay/samplematrixcode (英語)) に含まれています。

コンパイラー・ドキュメントにも追加情報があります。

## 構造体配列 (AoS) と配列構造体 (SoA)

プロセッサーは、メモリーが連続してストリームされる場合に最適に動作します。キャッシュラインのすべての要素を SIMD レジスターに移動すると非常に効率的です。連続したキャッシュラインにロードする場合も、プロセッサーは順番にプリフェッチできます。構造体配列 (AoS) では、次のようにデータが格納されます。

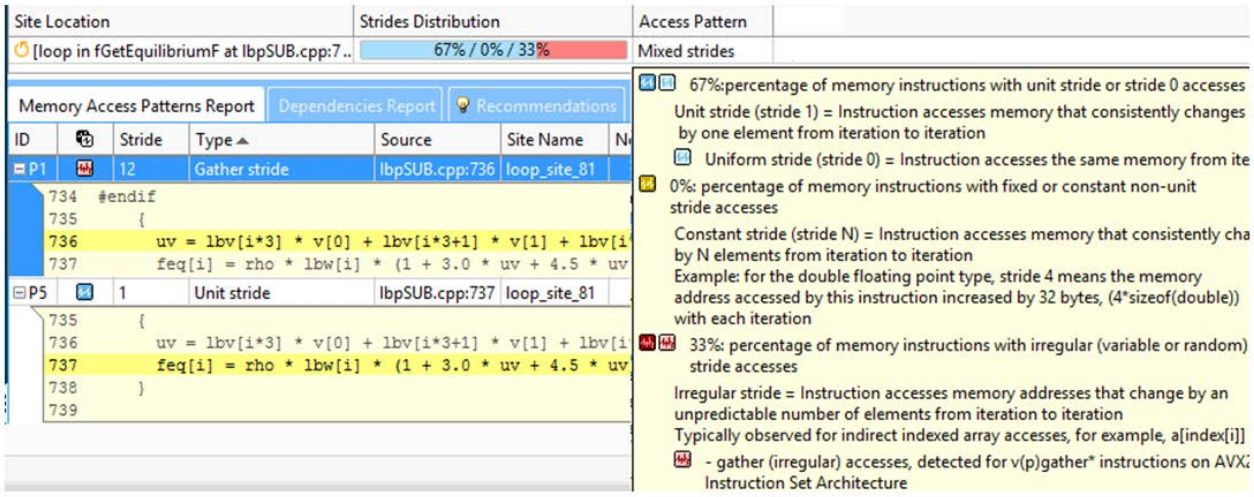
```
struct {
    uint r, g, b, w ; // 2D RGB ピクセルのレイアウト
} MyAoS[N] ;
```

このレイアウトでは、RGB 値が連続して配置されます。カラープレーンにあるデータを処理する場合、よく全体がキャッシュにロードされますが、一度に使用されるのは 1 つの値 (例えば g) のみです。配列構造体 (SoA) では、次のようにデータが格納されます。

```
struct {
    uint r[N] ;
    uint g[N] ;
    uint b[N] ;
    uint w[N] ;
} MySoA ;
```

データが配列構造体 (SoA) 形式で格納されている場合、キャッシュラインにデータがロードされると、キャッシュライン全体を使用してすべての g 値 (または r や b) を処理することができます。データを効率良く SIMD レジスターにロードできるため、効率とパフォーマンスが向上します。実際、必要に応じて、一時的にデータを配列構造体に移動して処理し、コピーバックすることがよくあります。この追加のコピーには時間がかかるため、できるだけ回避したほうがよいでしょう。

インテル® Advisor 2016 の Memory Access Pattern (MAP: メモリー・アクセス・パターン) 解析は、連続 ("ユニットストライド")、非連続、および "不規則な" アクセスパターンのループを識別します。




[Strides Distribution (ストライドの分布)] 列に、ソースループにおける各パターンの頻度の集約統計が示されます。上のスクリーンショットでは、バーの 2/3 が連続アクセスパターンを示す青色ですが、1/3 は非連続アクセスパターンを示す赤色です。純粋な AoS パターンを使用するコードでは、インテル® Advisor は AoS -> SoA 変換を実行するように特定の推奨事項を自動で示します。

インテル® Advisor の MAP では、アクセスパターンと一般的なメモリーの局所性解析が単純化されているだけでなく、メモリー・フットプリント・メトリックも提供され、各ストライド (アクセスパターン) が特定の C++ または Fortran オブジェクト名/配列名にマップされます。インテル® Advisor の詳細は、次の Web サイトを参照してください。

https://software.intel.com/en-us/get-started-with-advisor (英語)
https://www.isus.jp/intel-advisor-xe/

配列構造体と構造体配列データレイアウトは、多くのグラフィックス・プログラム、N 体 (分子動力学)、あるポイントや特定の物体に関連したデータ/プロパティー (質量、位置、速度、分量) で利用されます。一般に、配列構造体のほうが効率良く、優れたパフォーマンスをもたらします。

インテル® コンパイラー 16.0 Update 1 から、SIMD Data Layout Templates (SDLT) を利用することで AoS -> SoA 変換を簡単に行えるようになりました。SDLT を使用して、次のように AoS コンテナーを再定義します。

```
SDLT_PRIMITIVE(Point3s, x, y, z)
sdlt::soa1d_container<Point3s> inputDataSet(count);
```

これで、Point3s インスタンスに SoA 形式でアクセスできます。SDLT の詳細は、
https://software.intel.com/en-us/node/684050 (英語) を参照してください。

AoS と SoA に関する以下の記事もお読みになることをお勧めします。

https://www.isus.jp/products/psxe/mic-article/case-study-comparing-aos-and-soa/
https://software.intel.com/en-us/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture (英語)
http://stackoverflow.com/questions/17924705/structure-of-arrays-vs-array-of-structures-in-cuda (英語)

多くの場合、配列構造体のほうがアクセスパターンに一致し、優れたパフォーマンスを提供しますが、場合によっては構造体配列のほうがデータの参照と使用に適しており、パフォーマンスを向上できることがあります。

## まとめ

ここでは、データレイアウトとパフォーマンスについて考察する際の基本原則を示しました。コードは、データの移動が最小限になるように構成します。レジスターやキャッシュにあるデータを再利用します。これは、データの移動を最小限に抑えるのにも役立ちます。ループ・ブロッキングはデータの移動を最小限に抑えるのに有用です。特に、2D または 3D レイアウトを使用するソフトウェアでは効果があります。並列計算におけるタスクとデータの分配を考慮して並列処理のレイアウトを考えます。適切なドメイン分割により、メッセージパッシング (MPI) と共有メモリー・プログラミングの両方で利点が得られます。配列構造体は通常、構造体配列よりもデータ移動が少なく、優れたパフォーマンスをもたらします。フォルス・シェアリングを回避し、ローカル変数を作成するか、パディングを追加することで、各スレッドが異なるキャッシュライン上の値を参照するようにします。そして、キャッシュライン上でデータをアライメントします。

サンプルコードは、https://github.com/drmackay/samplematrixcode (英語) からダウンロードできます。

パート 1 はこちらから参照できます。

ここで紹介した手法を適用して、コードのパフォーマンスがどれぐらい向上するか試してみてください。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。